Calidad y Preprocesamiento de Datos

MCIC VÍCTOR MANUEL CORZA VARGAS

MC FERNANDO AVITÚA VARELA

Unidad 3 Perfilado de Datos

Objetivo

- Comprender y aplicar técnicas de perfilado de datos para evaluar la calidad de un conjunto de datos, detectando registros duplicados, datos incompletos, valores anómalos y otras dimensiones de calidad.

Introducción al Perfilado de Datos

¿Qué es el perfilado de datos?

- El perfilado de datos es el proceso de examinar y analizar datos para comprender su estructura, contenido y calidad. Es una etapa fundamental en la preparación de datos, permitiendo identificar problemas como valores faltantes, duplicados, errores y anomalías antes del análisis o integración.

Importancia del perfilado de datos

Mejora la calidad de datos: Permite detectar inconsistencias, valores atípicos y errores.

Facilita la limpieza y transformación de datos: Ayuda a definir reglas de preprocesamiento.

Optimiza la integración de datos: Esencial en la consolidación de múltiples fuentes de información.

Permite la toma de decisiones informadas: Datos de calidad mejoran la precisión de modelos analíticos y reportes.

Orden en el DAMA-DMBOK

El **perfilado de datos** (data profiling) es un paso previo a las estrategias de **calidad de datos** (data quality), porque permite descubrir problemas y definir estrategias antes de aplicar mejoras en la calidad de los datos.

Sin un perfilado adecuado, las iniciativas de calidad pueden ser ineficaces o basarse en suposiciones incorrectas.

Orden en el DAMA-DMBOK

Las métricas específicas son obtenidas del perfilado de datos.

Dimensión de Calidad	Métrica	Resultado	Interpretación
Precisión	% de datos incorrectos en nombres o direcciones	12%	12% de los registros tienen errores ortográficos o de formato
Consistencia	% de cuentas con datos incoherentes en distintos sistemas	18%	Datos no coinciden entre la base de clientes y el sistema de facturación
Completitud	% de registros con campos vacíos	9%	9% de los clientes no tienen número de teléfono registrado
Oportunidad	Tiempo promedio para actualizar datos incorrectos	48 horas	Se tarda hasta 2 días en corregir información tras detectar un error

Perfilado de datos

Orden en el DAMA-DMBOK

Data Profiling (Perfilado de datos)

- Se encuentra dentro de la función de Gestión de Calidad de Datos (Data Quality Management).
- Se lleva a cabo como una fase de diagnóstico para analizar la estructura, distribución y relaciones de los datos.
- Ayuda a identificar problemas como valores nulos, inconsistencias, duplicados y anomalías.

Data Quality Assessment & Improvement (Evaluación y mejora de la calidad de datos)

- Basándose en los hallazgos del perfilado de datos, se evalúan las dimensiones de calidad (Precisión, Completitud, Consistencia, Oportunidad, etc.).
- Se implementan estrategias para corregir errores y mejorar la confiabilidad de los datos.

Técnicas principales del perfilado de datos

- Estadísticas descriptivas: Análisis de frecuencias, medias, medianas, desviaciones estándar. (df.describe)
- Detección de duplicados: Identificación de registros repetidos.
- Detección de valores faltantes: Evaluación de la completitud de datos.
- Análisis de valores anómalos: Detección de outliers mediante histogramas y métodos avanzados.
- Medición de calidad en diferentes dimensiones: Evaluación de precisión, completitud, consistencia y oportunidad.

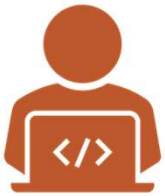
Detección y Medición de Registros Duplicados

Unicidad

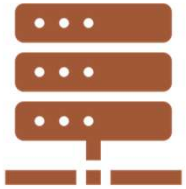
¿Qué es la unicidad en los datos?

La unicidad se refiere a que cada registro en un conjunto de datos representa una entidad única y no está repetido. Los registros duplicados pueden generar errores en los análisis, sesgar modelos y causar inconsistencias en bases de datos.

Causas comunes de duplicados



Errores humanos: Datos ingresados más de una vez manualmente.



Diferentes fuentes de datos: Fusiones de bases con redundancias.

Errores en procesos ETL: Integraciones mal diseñadas generan repeticiones.

Técnicas para detectar registros duplicados

Comparación exacta: Identificación de registros idénticos en todos los campos.

Detección de Duplicados Parciales

Comparación difusa (Fuzzy Matching): Uso de métricas como:

Distancia de Levenshtein: Mide la diferencia en caracteres entre dos cadenas.

Coeficiente de Jaccard: Evalúa la similitud entre conjuntos de palabras.

Algoritmo Soundex: Útil para nombres con variaciones ortográficas.

Comparación exacta de duplicados

```
import pandas as pd

data = {'ID': [1, 2, 2, 3, 5, 9], 'Nombre': ['Ana', 'Luis', 'Luis', 'Maria', 'Pablo', 'Pablo']}

df = pd.DataFrame(data)

# Detección de duplicados

duplicados = df.duplicated().sum()

print(f"\nNúmero de registros duplicados: {duplicados}")
```

Detección de Duplicados Parciales

A veces los registros tienen coincidencia en un subconjunto de columnas.

```
import pandas as pd
data = {'ID': [1, 2, 2, 3, 5, 9], 'Nombre': ['Ana', 'Luis', 'Luis', 'Maria', 'Pablo', 'Pablo']}
df = pd.DataFrame(data)
#Detección parcial de duplicados
duplicados_parciales = df.duplicated(subset=['Nombre']).sum()
print(f"\nNúmero de duplicados parciales por el campo nombre: {duplicados_parciales}")
```

Detección de Duplicados Parciales

En otras ocasiones los registros tienen pequeños cambios (espacios extras, diferencias en mayúsculas, errores tipográficos). Para identificar registros similares con pequeñas variaciones en los nombres o valores, podemos utilizar coincidencias difusas.

```
from fuzzywuzzy import fuzz, process

# Comparar dos cadenas de texto

print(fuzz.ratio("Luis", "Luis "))  # Maneja diferencias leves en los nombres

print(fuzz.ratio("Luis", "Luisillo"))  # Evalúa similitud en términos de caracteres
```

¿A qué se le llama comparación difusa?

Los algoritmos como la distancia de Levenshtein, el coeficiente de Jaccard, el algoritmo soundex, entre otros, son conocidos como algoritmos de comparación difusa porque no realizan comparaciones estrictas o exactas, sino que miden el grado de similitud entre dos elementos (como cadenas de texto, conjuntos o vectores) en lugar de verificar una coincidencia binaria (igual o diferente).

Distancia de Levenshtein

La **distancia de Levenshtein** mide el número mínimo de operaciones necesarias para transformar una cadena en otra. Las operaciones permitidas son:

1. **Inserción**: Agregar un carácter.
2. **Eliminación**: Quitar un carácter.
3. **Sustitución**: Cambiar un carácter por otro.

Distancia de Levenshtein

Ejemplo: Convertir "gato" en "pato"

La palabra `"gato"` se puede convertir en `"pato"` en un solo paso:

- Sustituimos la `"g"` por `"p"`.

Distancia de Levenshtein = 1

Distancia de Levenshtein

```
import Levenshtein

# Distancia entre dos palabras
print(Levenshtein.distance("gato", "pato"))  # Salida: 1
print(Levenshtein.distance("silla", "salón"))  # Salida: 3
print(Levenshtein.distance("casa", "caso"))  # Salida: 1
```

Distancia de Levenshtein

Explicación del ejemplo 2:

Queremos transformar "silla" en "salón":

1. **Sustituir** "i" → "a" resultado: salla
2. **Sustituir** segunda "l" por "ó" resultado: saláa
3. **Sustituir** "a" por "n" al final → "salón"

Distancia de Levenshtein = 3

Distancia de Levenshtein

¿Porqué cuando utilizamos este código, el resultado es 89?

```
from fuzzywuzzy import fuzz, process

# Comparar dos cadenas de texto
print(fuzz.ratio("Luis", "Luis "))  # Maneja diferencias leves en los nombres
```

Distancia de Levenshtein

Similitud

Levenshtein cuenta el número de ediciones (inserciones, eliminaciones o sustituciones) necesarias para convertir una cadena en otra. Siendo el resultado un número entero. Lo que representa un problema, ya que el número de errores que se pueden esperar en una cadena es proporcional a la longitud de esta. Un problema que se puede solucionar normalizando la distancia para obtener el grado de similitud:

$$1 - (lev_{a,b}(|a|, |b|)/max(|a|, |b|))$$

Distancia de Levenshtein

El valor **89** en la comparación `fuzz.ratio("Luis", "Luis ")` usando `fuzzywuzzy` se debe a que la biblioteca usa la **distancia de Levenshtein normalizada** en porcentaje, y la diferencia entre las cadenas es un espacio adicional al final.

- Para transformar `"Luis"` en `"Luis "`, se necesita **una inserción** (añadir el espacio).
- Longitud total de la cadena más larga: **5** (porque `"Luis "` tiene 5 caracteres).

Ratio = (1 – (Distancia de Levenshtein / Longitud de la cadena más larga) × 100

Ratio = (1 – (1 / 5)) x 100

Ratio = 80% (La diferencia se debe a la implementación de fuzzywuzzy, puede utilizar la distancia de python-Levenshtain o difflib.SequenceMatcher).

Sequence Matcher

SequenceMatcher.ratio() se calcula así:

ratio = 2 . M / |a|+|b|

M es el número de caracteres que coinciden y pueden emparejarse en orden en ambas cadenas. Es decir: cuántos caracteres ‘iguales’ forman la parte común entre los dos textos.

Sequence Matcher

- `a = "Luis"` → |a|=4
- `b = "Luis "` → |b|=5 (el espacio cuenta)
- Los bloques coincidentes emparejan `"Luis"` completo → M=4

Entonces:

$2 \cdot 4 / 4 + 5 = 8/9 = 0.888\ldots \Rightarrow 88.8\%$

Coeficiente de Jaccard

El coeficiente de Jaccard es una métrica utilizada para medir la similitud entre dos conjuntos. Cuando se aplica a cadenas de texto, se basa en la comparación de los conjuntos de n-gramas de cada cadena.

Definición formal

Dado dos conjuntos A y B, el coeficiente de Jaccard se define como:

$J(A,B)= |A \cap B| / |A \cup B|$

donde:

$|A \cap B|$ es el número de elementos en la intersección de los conjuntos A y B.

$|A \cup B|$ es el número de elementos en la unión de los conjuntos A y B.

El resultado varía entre 0 (sin similitud) y 1 (idénticos).

Coeficiente de Jaccard

Aplicación a cadenas de texto

Para aplicar esta métrica a cadenas de caracteres, se deben convertir en conjuntos de n-gramas (subcadenas de longitud n).

Ejemplo

Supongamos que tenemos dos cadenas:

$S1$ = "casa"

S2 = "caso"

Si tomamos bigramas (n=2), los conjuntos serían:

S1 → { "ca", "as", "sa" }

S2 → { "ca", "as", "so" }

Coeficiente de Jaccard

Ahora, calculamos:

- **Intersección**: { "ca", "as" } → tiene 2 elementos.
- **Unión**: { "ca", "as", "sa", "so" } → tiene 4 elementos.

Por lo tanto, el coeficiente de Jaccard es:

$J(S1,S2)=2/4=0.5$

Algoritmos Fonéticos

Son algoritmos diseñados para comparar palabras según su sonido en lugar de su escritura.

Se utilizan en sistemas de búsqueda aproximada, corrección ortográfica y genealogía.

Ayudan a encontrar palabras con pronunciación similar aunque estén escritas de forma diferente.

Algoritmo Soundex

Convierte palabras en un código de 4 caracteres (letra + 3 números).

Reglas:

1. Se conserva la primera letra de la palabra.
2. Se asignan números a las consonantes (Ej: B, F, P, V = 1; C, G, J, K, Q, S, X, Z = 2, etc.).
3. Vocales y algunas consonantes se eliminan.
4. Se rellena o trunca a 4 caracteres.

Ejemplo: "Robert" y "Rupert" tienen el mismo código (R163).

Metaphone

- Mejora Soundex al basarse en reglas fonéticas en inglés.
- Genera claves fonéticas en vez de códigos numéricos.

Metaphone:

- Introducido por Lawrence Philips en 1990.
- Considera reglas de pronunciación en inglés.

Double Metaphone:

- Segunda versión que genera dos claves por palabra (pronunciación primaria y alternativa).
- Mejor precisión con nombres y palabras extranjeras.

Otros Algoritmos Fonéticos

NYSIIS:

- Desarrollado por el estado de Nueva York.
- Mantiene la estructura original de la palabra mejor que Soundex.

Caverphone:

- Diseñado en Nueva Zelanda para nombres propios.
- Se usa para genealogía y bases de datos históricas.

Cologne Phonetic:

- Utilizado en Alemania para comparar nombres fonéticamente.
- Similar a Soundex pero optimizado para el idioma alemán.